RCP Database Interactions

Version 2.3

This document presents the database interface of the **RCP** system. The interface is used by all RCP database implementations.

RCP Database Interactions

Version 2.3

Marc Paterno
University of Rochester

KEY TO RECENT CHANGES

Version 2.3

1. Removed `RCPValue::fillFrom(const RCPScript& v)` because it introduced unwanted coupling. This behavior will now belong to **RCPScript**.
2. Minor changes to explanation of `put(RCPValue& val)` in both **RCPDatabaseServices** and **AbsRCPDatabase**.
3. Change from `std::string` to `const char *` in required argument for constructor for classes inheriting from **AbsRCPDatabase**.
4. All "counting" functions return `size_t`, rather than `int`.
5. All functions in **AbsRCPDatabase** are now pure virtual functions.
6. Made **RCPValue** contain a `std::list` of all relevant **RCPNames**, rather than a single name; added `size_t numNames() const` function to **RCPValue**; replaced `RCPName name() const` with `void names(RCPNameList& names) const`.
7. Add function `virtual bool isWriteable() const = 0` to the interface of **AbsRCPDatabase**.
8. Add `remove`... functions to **RCPValue**.

Version 2.2

1. Addition of extra functions to **RCPValue** interface, allowing queries of the number of each type of contained object, and the names of the contained objects.

INTRODUCTION AND TERMINOLOGY

This document specifies the interactions of the RCP system with all RCP database implementations.

In the following, we carefully distinguish between RCPs (instances of the class **RCP**); RCPValues (instances of the class **RCPValue**), RCP scripts (text files used to describe an RCP) and RCP database entries (the representations of an RCP objects in a specific physical database).

Glossary

The following list of terms is presented in order

Parameter set: the collection of name/value pairs in an **RCPValue** object.

RCPName: a set of four strings: package name, object name, version, and database name.

RCPID: a unique identifier for a parameter set.

RCPValue: the object containing a parameter set, and **RCPID**, and an **RCPName**.

RCP: an **RCPValue** object, wrapped in a read-only shell. This is form in which parameter sets are presented to users.

RCPHashKey: a not-quite-unique identifier, calculated by an **RCPValue**, from the contents of its parameter set.

Database piece: group of tables in a database, which contain related parameter **RCPValue** objects. Different database pieces may or may not reside in different physical databases.

Concrete database object: an instance of a subclass of **AbsRCPDatabase**.

RCP Classes Relevant to the Databases

The database interactions of the RCP package are encapsulated in a single abstract base class, **AbsRCPDatabase,** from which concrete RCP database classes are derived by subclassing. These subclasses are used, through the **AbsRCPDatabase** interface, by the class **RCPDatabaseServices**. The RCP database objects will be constructed by a factory on behalf of the **RCPManager,** which is responsible for managing all interactions with the physical database represented, within a program, by an RCP database object. Users of the RCP system do not interact directly with the database.

The purpose of the **RCPDatabaseServices** class is to provide the database services needed by the **RCPManager**, through a simple interface. The purpose of the **AbsRCPDatabase** class is to simplify the job of the concrete subclasses, by factoring out the behavior common to all such subclasses.

The things manipulated by the **AbsRCPDatabase** interface are **RCPValue** objects. An **RCPValue** object contains the data for a single **RCP** object and an interface to query and modify the contained data.

Each RCP database entry is associated with a unique **RCPID** identifier. Each *distinct* physical database is associated with a unique database identifier, which is also contained in the **RCPID**. We note that a copy of a database is not distinct from the master from which it was made because the copy can be used *only* in read-only mode. Write access is available only for the master version. A **RCPHashKey** represents the “hash value” associated with an RCPValue object. This value will be used as a non-unique (but as close to unique as can reasonably be achieved) key to find RCP database entries. The **RCPHashKey** associated with an **RCPValue** can be calculated from the **RCPValue** object alone, using only the (name, value) pairs it contains, and *not* using the associated **RCPID**. **RCPHashKey** is currently a typdef for unsigned int — a 16-bit CRC. It has been made a typedef so that this may be changed, if necessary, with little effort.

PROGRAMMATIC REPRESENTATION OF DATABASES AND PARTS OF DATABASES

The representation of a physical database, or of a portion of a database, to the RCP system is via an instance of a subclass of **AbsRCPDatabase** — a *concrete RCP database object*. It is critical to the functioning of the RCP system that the correct association be made between each concrete RCP database object and the physical database, or portion thereof, it represents. It is simplest to explain the requirements through an example.

Consider a single Oracle database containing all the RCP database entries for DØ. The tables in this database are organized into groups.

- One group of tables for officially released RCPs — named "official".
- Two groups of tables, one for the Higgs physics group and one for the QCD physics group — named "higgs" and "qcd".
- Two groups of tables, one for each of two users, Jack and Jill — named "jack" and "jill".

We shall call each of these groups of tables a *database piece*. Each database piece has an associated name (a string), and an associated **RCPDatabaseID**. The master Oracle database is responsible for issuing unique names and **RCPDatabaseID**s for database pieces.

In any program, we want to make sure that we have only one programmatic representation of each database piece. So, when we create an instance of **OracleRCPDatabase** (a subclass of **AbsRCPDatabase**), we want to connect it with a specific database piece, which we specify by giving the name of the database piece to the constructor, as follows:

```
OracleRCPDatabase jacksDB("jack");
```

The RCP system will make use of this constructor to assure that all **RCPManager**s in a single program that want to talk to a specific database piece do so through the same concrete database object.

RESPONSIBILITIES OF RCPDATABASESERVICES AND ABSRCPDATABASE

The **RCPDatabaseServices** class exists to provide a common implementation of the functions related specifically to the behavior required of the RCP system. The **AbsRCPDatabase** classes, and its concrete subclasses, are to perform only the database-specific (including the client/server nature) parts of this behavior.

To be more specific:

- **RCPDatabaseServices** provides caching of **RCPValue** objects that are extracted from the **AbsRCPDatabase** object. A concrete database class does not have to implement this caching.
- When an **RCPManager** requests an **RCPValue** that matches the contents of a given **RCPValue** (by supplying a complete **RCPName** and a valid **RCPID**), it is the **RCPDatabaseServices** class that determines which of the possible matches, if any, is the correct one. The **AbsRCPDatabase** class is responsible only for (1) telling how many parameter sets match a given hash key, and (2) returning all the parameter sets matching that has key.

- A concrete subclass (or subclasses) of **AbsRCPDatabase** is responsible for providing the client/server nature of the database connection. An **RCPDatabaseServices** object talks only to an instance of a subclass of **AbsRCPDatabase** which exists in the same process. The details of how the client/server implementation is done is up to the implementer of the concrete **AbsRCPDatabase** subclass.

UNIQUENESS OF RCPNAMES AND RCPIDS

The class **RCPName** exists in order to provide a human-friendly method of referring to a particular parameter set. The class **RCPID** exists in order to provide a concise and unique method of referring to a particular parameter set.

The mapping from **RCPID** to parameter set is *one-to-one*: each parameter set has exactly one **RCPID**, and each **RCPID** refers to a one parameter set.

The mapping of **RCPName** to parameter set is *many-to-one*: each parameter set can have many names (it must have at least one), but each **RCPName** must refer to a single parameter set.

An **RCPName** object has several components, each of which is a string: a *package name*, and *object name*, a *version*, and a *database name*. For parameter sets which are entered into the database through the release mechanism, the *version* means the version name of the DØ software release. For parameter sets entered by any other mechanism (such as a WWW interface), the assignment of the version name will be done by the database. The *database name* is the name associated with the database piece in which the parameter set resides.

typedefs

These are the typedefs defined and used by the RCP system.

```
typedef unsigned int RCPHashKey;

typedef unsigned int RCPDatabaseID;

typedef std::list<RCPValue> RCPValueCollection;

typedef std::list<RCPName> RCPNameList;
```

RCPValue

The **RCPValue** class is the programmatic view of a parameter set. It provides methods to access and add to the values in the parameter set. It also carries the associated **RCPID** and **RCPName** objects.

Memory Management

*Create an empty **RCPValue**, with an invalid **RCPID** and an empty **RCPNameList.***

```
RCPValue( );
```

*Create an **RCPValue** with the given **RCPID**, and and empty **RCPNameList**.*

```
RCPValue(const edm::RCPID& id);
```

Copy constructor.

```
RCPValue(const RCPValue& rhs);
```

Assignment operator.

```
RCPValue& operator=(const RCPValue& rhs);
```

Destructor.

```
~RCPValue( );
```

Testing

Equality test compares for equality of contained parameter sets.

```
bool operator== (const RCPValue& rhs) const;
```

Helper function used by equality test operator.

```
bool sameContentsAs(const RCPValue& val) const;
```

Negation of operator==().

```
bool operator!=(const RCPValue& rhs) const;
```

*Return true if the **RCPValue** contains any value with the given name.*

```
bool containsName(std::string name) const;
```

*Each of the following tests returns true if the **RCPValue** contains a value of the given type with the given name.*

```
bool containsBool(std::string name) const;

bool containsInt(std::string name) const;

bool containsFloat(std::string name) const;
```

```
bool containsString(std::string name) const;
bool containsRCPID(std::string name) const;
bool containsVBool(std::string name) const;
bool containsVInt(std::string name) const;
bool containsVFloat(std::string name) const;
bool containsVString(std::string name) const;
bool containsVRCPID(std::string name) const;
```

Manipulation and access

*Each get… function either returns the value matching the name, or, if none matches, throws an **XRCPNotFound** exception. Each add… function first tests to see whether any value (not just a value of the given type) with the given name is already stored; of so, an **XRCPDuplicate** exception is thrown. If not, then the given value is stored with the given name.Each remove… function removes a value of the given name and type, and returns true; if no value of the given name and type is found, it returns false without modifying the contents.*

```
bool getBool(std::string name) const;
void addBool(std::string name, bool value);
bool removeBool(std::string name);
std::vector<bool> getVBool(std::string name) const; bool
void addVBool(std::string name, const std::vector<bool>& value);
bool removeBool(std::string name);
int getInt(std::string name) const;
void addInt(std::string name, int value);
bool removeInt(std::string name);
std::vector<int> getVInt(std::string name) const;
void addVInt(std::string name, const std::vector<int>& value);
bool removeVInt(std::string name);
float getFloat(std::string name) const;
void addFloat(std::string name, float value);
bool removeFloat(std::string name);
std::vector<float> getVFloat(std::string name) const;
void addVFloat(std::string name,
   const std::vector<float>& value);
bool removeVFloat(std::string name);
std::string getString(std::string name) const;
void addString(std::string name, std::string value);
bool removeString(std::string name);
```

```
std::vector<std::string> getVString(std::string name) const;

void addVString(std::string name,
   const std::vector<std::string>& value);

bool removeVString(std::string name);

edm::RCPID getRCPID(std::string name) const;

void addRCPID(std::string name, const edm::RCPID& value);

bool removeRCPID(std::string name);

std::vector<edm::RCPID> getVRCPID(std::string name) const;

void addVRCPID(std::string name,
   const std::vector<edm::RCPID>& value);

bool removeVRCPID(std::string name);
```

*Return the **RCPID** that identifies this **RCPValue**.*

```
edm::RCPID myRCPID( ) const;
```

*Set the **RCPID** for this **RCPValue**. The precondition is that the given **RCPID** be valid.*

```
void setRCPID(const edm::RCPID& id);
```

*Get the **RCPNames** for this **RCPValue**.*

```
void names(RCPNameList& names) const;
```

*Set the **RCPNames** for this **RCPValue**. The precondition is that each name in the given **RCPNameList** be complete.*

```
void setNames(const RCPNameList& names);
```

*Add the given **RCPName** to those for this **RCPValue**. The precondition is that the new **RCPName** must be complete.*

```
void addName(const RCPName& name);
```

*Return the hash key for the parameter set contained in this **RCPValue**.*

```
RCPHashKey hash( ) const;
```

Helper function used to compute the hash value for the contained parameter set.

```
RCPHashKey computeHash() const;
```

Return the number of parameters of each type stored in the parameter set.

```
size_t numBools() const;

size_t numVBools() const;

size_t numInts() const;

size_t numVInts() const;

size_t numFloats() const;

size_t numVFloats() const;

size_t numStrings() const;

size_t numVStrings() const;

size_t numRCPIDs() const;

size_t numVRCPIDs() const;
```

Return the total number of parameters stored in the parameter set.

```
size_t numEntries() const;
```

Fill the vector names with the names of all parameters of the given type in the parameter set.

```
void getBoolNames(std::vector<std::string>& names) const;

void getVBoolNames(std::vector<std::string>& names) const;

void getIntNames(std::vector<std::string>& names) const;

void getVIntNames(std::vector<std::string>& names) const;

void getFloatNames(std::vector<std::string>& names) const;

void getVFloatNames(std::vector<std::string>& names) const;

void getStringNames(std::vector<std::string>& names) const;

void getVStringNames(std::vector<std::string>& names) const;

void getRCPIDNames(std::vector<std::string>& names) const;

void getVRCPIDNames(std::vector<std::string>& names) const;
```

*Fill the vector names with the names of all the parameters within this **RCPValue** object.*

```
void getNames(std::vector<std::string>& names) const;
```

Output

Print output to stream os, in a form useful for debugging.

```
void dump(std::ostream& os) const;
```

Print output to stream os, in a form useful for display of the contents.

```
friend std::ostream& operator<<(std::ostream& os,
   const RCPValue& r);
```

RCPName

Each **RCPName** maps to exactly one parameter set, but one parameter set may be associated with more than one **RCPName**.

Memory management

Constructor.

```
RCPName(std::string pkgname, std::string objname,
   std::string release = "", std::string dbname = "");
```

Testing

*Return true if this **RCPName** has all of (pkgname, objname, release, dbname) as non-empty strings; return false if one or more is an empty string.*

```
bool isComplete( ) const;
```

Manipulation and access

```
std::string pkgName( ) const;

void setPkgName(std::string pkgname);

std::string objName( ) const;

void setObjName(std::string objname);
```

```
std::string release( ) const;
void setRelease(std::string release);
std::string dbName( ) const;
void setDBName(std::string dbname);
```

RCPID

Each **RCPID** is the unique identifier for a parameter set.

Memory management

*Create an invalid **RCPID**.*

```
RCPID( );
```

*Create a valid **RCPID**.*

```
RCPID(unsigned int serialnumber,
   const RCPDatabaseID& databaseID);
```

Testing

```
bool isValid( ) const;
```

*Return true if this **RCPID** was issued by the "official" database piece.*

```
bool isOfficial( ) const;
```

RCPDatabaseServices

This class is the interface to be used by all clients requiring the services of an RCP database. One instance of **RCPDatabaseServices** is associated with one database piece, as described above. **RCPDatabaseServices** collaborates with **RCPManager** and shares the knowledge of how parameter sets are managed. This class relies on an instance of a concrete subclass of **AbsRCPDatabase** to perform actual communication with an external database.

Testing

Return true if the db contains a parameter set with this id.

```
bool has(const edm::RCPID& id);
```

Return true if the db contains a parameter set which matches that of val.

```
bool has(const RCPValue& val);
```

Return the number of parameter sets matching this (possibly incomplete) name.

```
size_t count(const RCPName& name);
```

Access

*If the db contains a parameter set with this ID, set the contents of val to match, and return true. If the db does not contain a match, do not modify val, and return false. If this function returns true, then val is assured to have a complete **RCPName** and a valid **RCPID**.*

```
bool get(const edm::RCPID& id, RCPValue& val);
```

*Fill the collection with all those **RCPValue**s that match the given (possibly incomplete) **RCPName**, and return true. This requires (1) making default (empty) **RCPValues**, (2) setting their contents with the various **add…** functions, and (3) inserting the appropriate **RCPID**s and **RCPName**s. If no matching **RCPValue**s are found, return false and do not modify the collection values.*

```
bool get(const RCPName& name, RCPValueCollection& values);
```

*If possible, complete the given **RCPValue**. If an entry in the db has a parameter set matching that of val, then set the **RCPName** and **RCPID** of val to be the same as that of the entry in the db and return true. If no entry in the db has a parameter set matching that of val, return false and do not modify val. Note that the determination of matching contents makes use of the hash key, which val can generate, and so an exhaustive test for equality is required only when the hash keys are equal. Note also that **RCPValue** has an equality test function that is used to perform the equality test. This equality test makes use of the hash key test, so it is not necessary to perform this test before testing two **RCPValue**s for equality.*

```
bool get(RCPValue& val);
```

*Add a parameter set equal to the one within val to the db. Modify val to have a valid **RCPID** (issued by the db, to assure uniqueness) and to have a complete **RCPName**, and return true. The **RCPName** may have to be completed by the db by inserting the database name, and perhaps a version, again to assure uniqueness. If the new parameter set cannot be added to the database, or if a new unique **RCPID** cannot be issued, or if the **RCPName** cannot be completed uniquely, return false and do not modify val.*

```
bool put(RCPValue& val);
```

AbsRCPDatabase

This is an abstract class that defines the interface for all concrete database objects. Any client/server behavior of the database connection must be implemented in subclasses of this class.

Each class that inherits from **AbsRCPDatabase** is required to have a single-argument constructor that takes a `const char *`; this string is used to determine which database piece is represented (and communicated with) by the constructed object.

Memory management

Destructor

```
virtual ~AbsRCPDatabase( ) = 0;
```

Testing

Return true if the db contains an entry with this ID.

```
virtual bool has(const edm::RCPID& id) const = 0;
```

Return the number of db entries matching this name. Note that the name may be incomplete; this is why more than one match is possible.

```
virtual size_t count (const RCPName& name) const = 0;
```

Return the number of db entries matching this hash key.

```
virtual size_t count(const RCPHashKey& key) const;
```

Return true if writing to this db is allowed.

```
virtual bool isWriteable( ) const = 0;
```

Manipulation and access

*Fill the given **RCPValue** with the parameter set specified by the given **RCPID**, insert the appropriate **RCPID** and **RCPName,** and return true. If no match is found, return false and do not modify val.*

```
virtual bool get(const edm::RCPID& id,
   RCPValue& val) const = 0;
```

*Fill the collection with **RCPValues** that match the given hash key, and return true. This requires (1) making default (empty) **RCPValues**, (2) setting their contents with the various **add…** functions, and (3) inserting the appropriate **RCPID**s and **RCPName**s. If no matching **RCPValue**s are found, return false and do not modify the collection values.*

```
virtual bool get(const RCPHashKey& key,
   RCPValueCollection& values) const = 0;
```

*Fill the collection with **RCPValue**s that match the given (possibly incomplete) **RCPName**. This requires (1) making default (empty) **RCPValues**, (2) setting their contents with the various **add…** functions, and (3) inserting the appropriate **RCPID**s and **RCPName**s. If no matching **RCPValue**s are found, return false and do not modify the collection values.*

```
virtual bool get(const RCPName& name,
   RCPValueCollection& values) const = 0;
```

*Add a parameter set equal to the one within val to the db. Modify val to have a valid **RCPID** (issued by the db, to assure uniqueness) and to have a complete RCPName, and return true. The **RCPName** may have to be completed by the db, by inserting the database name, again to assure uniqueness. If the new parameter set cannot be added to the database, or if a new unique **RCPID** cannot be issued, or if the **RCPName** cannot be completed uniquely, return false and do not modify val.*

```
virtual bool put(RCPValue& val) = 0;
```

Output

Print output useful for debugging to stream os.

```
virtual void dump(std::ostream& os) const = 0;
```